VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Tuomo Soppela

DEVELOPING CONTAINERIZED MICROSERVICES

Fliq Oy

Liiketalous
2020

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietojenkäsittelyn koulutusohjelma

ABSTRACT

Author	Tuomo Soppela
Title	Developing Containerized Microservices
Year	2020
Language	English
Pages	62
Name of Supervisor	Raija Tuomaala

The goal of this thesis is to research and evaluate the central processes of developing containerized microservice applications based on the employer Fliq Oy's current web application.

The aim is to cover the basic theory and the most widely used technologies in microservice development and how they come together in deploying finalized software applications.

The act of moving from a traditional application architecture into a microservice model is a complex process that involves making changes at every step of the development and deployment lifecycle. This thesis focuses on moving from a traditional application infrastructure to a containerized one and does not attempt to recreate the entire application development process from scratch.

Keywords Microservice, container, Docker, Kubernetes

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

1.1 Employer

Fliq Oy is a software developer specializing in smart factory solutions for several industrial clients centered around Ostrobothnia, Finland. The company was officially founded in Vasa, Finland on the 16th of August 2013 and currently employs about a dozen developers working on everything from web applications to desktop and mobile. Fliq Oy's primary product is the web-based version of their namesake application Fliq.

Fliq (the application) offers a factory monitoring and production visualization dashboard with options to observe and control processes from supply chains to factory production and project management. The dashboard is made up of modules, each with a designated purpose, such as worktime management, order tracking etc. The web dashboard is the central interface for the entire system, comprising of IoT sensor data as well as other features, such as mobile app integration.

Not unlike many other software startups today, Fliq Oy's initial product offering was built up as a monolithic web application, utilizing only a handful of technologies. The application was developed using XAMP, the Windows equivalent of the popular LAMP stack (Linux OS, Apache2 webserver, MySQL database and PHP). As the scope and requirements of their project has grown, Fliq Oy has come to face the challenges of maintaining a monolithic application and looks towards migrating to microservices for a solution.

All references related to Fliq Oy's customers, domains, URLs, databases, keys, certificates, login credentials, and application structure have been either redacted or altered where possible so as to not reveal any sensitive information about the employer.

1.2 Goal

The goal of this thesis is to research and test methods of clustering existing monolithic application structure into containerized microservices starting with the employer's current web application. The end goal was not to deliver a fully functional, production-ready containerized microservice application, but rather a small-scale demo project along with a well-documented framework of methods and technologies on how such an end goal could be accomplished. This thesis project was agreed to last a total of 3 months from the 3rd of February 2020 until the 30th of April in the same year.

1.3 Structure

Chapter 2: Requirements and Analysis covers the situation at the start of the project. The short chapter sets up a list of necessary steps to take within the scope of this project and lays out a roadmap for the thesis.

Chapter 3: Containerized Microservices goes over the relevant technologies associated with modern microservice architecture development. The list is not exhaustive as the complete assortment of included technologies would not fit into the length and time requirements set for this thesis. Rather, it focuses on the most central tools that serve a central purpose in developing containerized applications.

All the references to outside sources in this paper are made in this chapter. The reference material is restricted to the web pages of the official documentation of each relevant technology that is covered. The reasoning behind this is that in all cases at the time of writing, the official documentation is the most reliable and up-to-date source of information available and will continue to be so with a high degree of certainty.

Chapter 4: Development Process is the core of the thesis; Covering the development of the demo project step-by-step. The chapter will demonstrate the real-world use cases of the various technologies outlined in the previous chapter with a focus on some of the key challenges and how they were ultimately overcome.

Chapter 5: Conclusion details the state of the finished demo project and how it met the objectives set for it in the beginning. This final chapter also covers the next crucial steps in order to continue upon the work that was started during this project.

2 THE REQUIREMENTS ANALYSIS

2.1 Requirements

The initial requirements were laid out by the employer at the beginning of the project. As I had already interned at the company previously, I had worked on researching and evaluating backend technology alternatives to the current PHP backend. During my first internship, the development director at Fliq Oy ended up choosing Go-based containerized microservice architecture as the new application model to replace the aging PHP backend infrastructure. As such I already had some familiarity with the involved technologies going into the project.

As the employer's new backend development was already underway, I was initially tasked with containerizing the current state of the web application with the finished Go authentication microservice included with means to redirect incoming client requests to the PHP backend. The requests should be handled end-to-end, meaning that they would be processed in the backend code, which executes one or multiple database queries and then returns some data to the client. The web client is typically a web browser that renders the UI and sends the backend requests.

This end-to-end communication should then be SSL-encrypted over HTTPS using Let's Encrypt certificate authority issued certificates, the requesting and renewal of which should be automated using the popular certificate automation tool Certbot. All this should be done in Docker containers, following proven industry standards and best practices, and adhering to modern microservice architecture development. This meant the separation of logic into isolated and maintainable containers that can communicate between one another using Docker's integrated networking and the web application's REST API.

The microservice application would be developed purely in Docker at first and be likely deployed using an orchestration tool later. This should be taken into consideration early on, as all technologies would need to be able to accommodate the requirements set by container orchestration technologies.

2.2 Analysis

From the requirements set, the following containers are necessary:

- **Nginx Webserver**

  The Nginx container will be acting as both gateway and a reverse proxy, meaning it is the only service exposed to the client. The webserver will be responsible for handling incoming requests, redirecting them to their correct endpoints inside the application and returning responses back to the client. This means it should handle TLS handshakes and serve the UI files. Nginx is a popular choice for acting as a reverse proxy in front of microservice applications due to its simple yet flexible upstream virtual host configuration options. In the future, Nginx may also be configured to act as an ingress, a specialized load-balancer for Kubernetes deployments.
- **PHP backend API**

  The PHP backend container will include the current application backend in its entirety. Apache2 webserver is also included as PHP requires an external web server. Modelled as close to the original backend for compatibility, this container will eventually be deprecated as the new backend is completed and thus will be subject to less restrictions than the other containers.
- **Go backend API**

  The Go backend container will run the one existing Go authentication service. Go programs can be compiled into binary executables and served from their own internal HTTP web server so the resulting image should be quite compact. This container will probably serve as a reference in the future for other similar Go services.
- **Mariadb Database**

  The database container that is responsible for hosting the main Mariadb database used by the application. The database data must be stored in a persistent location. The database client also needs to be accessible from outside the application for any potential maintenance or scaling operations.

3 CONTAINERIZED MICROSERVICES

The virtualization of processes is growing rapidly in the world of software as emergent development and deployment technologies have matured and been welcomed into many software companies world-wide. Coupled with the ever-increasing popularity of cloud computing platforms, this have given rise to an all new manner of designing and maintaining applications, commonly referred to as microservice architecture.

Microservice architecture focuses on dividing an application into smaller components called microservices and then connecting them to one another to make up the entire application. This is in contrast with the traditional development model where every part of the application is built into a large, self-contained system without modularity. This model is referred to as a monolithic application.

The process of factoring applications into component parts is nothing new, however. Typically, more complex applications have been divided into a multi-tiered structure based on business logic; a front-end user interface connected to a back-end database through a middle-tier programming logic. As applications evolve over time by adapting to the customers' needs, even these kinds of monolithic applications often tend to inflate as new features and dependencies are integrated. Often this means that more resources need to be allocated to development and the process is slowed down. The technologies used to create the initial application may no longer accommodate the changing needs and requirements and a single error virtually anywhere in the system may cause everything to break down.

These are the kinds of problems that microservices attempt to address. To separate unique business logic into individual services that are entirely self-contained and modular from the rest of the application. Containers are an integral part of the microservice architecture. The aim of this thesis is to cover how the different technologies and design principles come together in developing and deploying modern containerized microservice-based software applications.

3.1 Docker

Docker container technology was first launched in 2013 as the open source Docker Engine written natively in Go programming language. Since then, the project has evolved into a robust service platform with both free community and paid enterprise editions with support across multiple operating systems including Windows, MacOS as well as a multitude of Linux distributions (Docker Inc. 2020a). Although other competing containerization alternatives have emerged since then, today Docker is considered to be the de-facto containerization solution in software development.

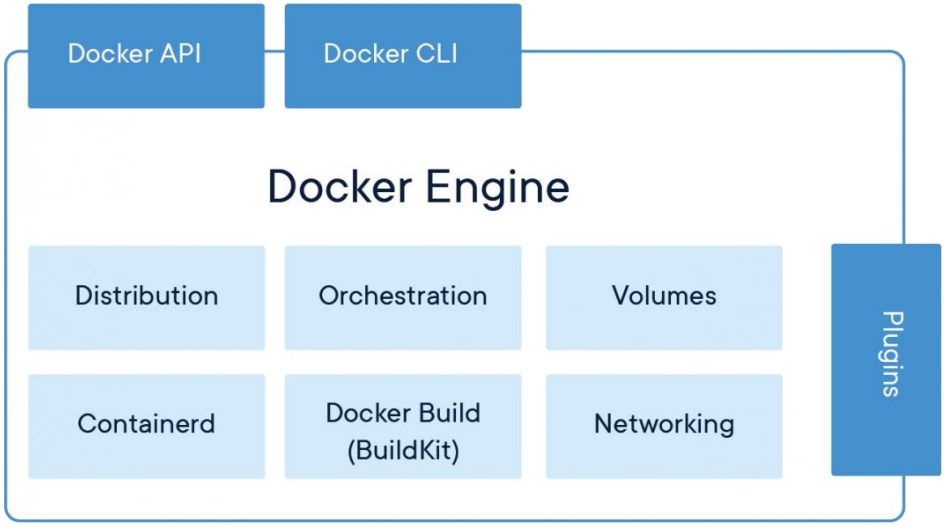


Figure 1. Docker Engine overview (Docker Inc. 2020b).

Docker can be divided into several tools and services that make up the platform. Docker client includes the command line tools and interfaces that hook up to the Docker daemon over TCP networking via an intermediate API. This implementation follows the client-server model with the client sending requests over a common network protocol and the server returning a response. The daemon is the server where most of the program logic takes place (Docker Inc. 2020c). The Client also enables interaction between the local daemon and a remote Docker registry where images can be stored for distribution. Unlike the daemon, registries are also capable of responding to image pull requests sent using HTTP. Docker Hub

is the default registry where all official images are hosted publicly for anyone to use.

The platform's architecture is based on the prevalent client-server model, which enables the client to talk to the daemon either locally or remotely. Docker references other web development conventions as well by utilizing a built-in REST API as the intermediate communication layer.

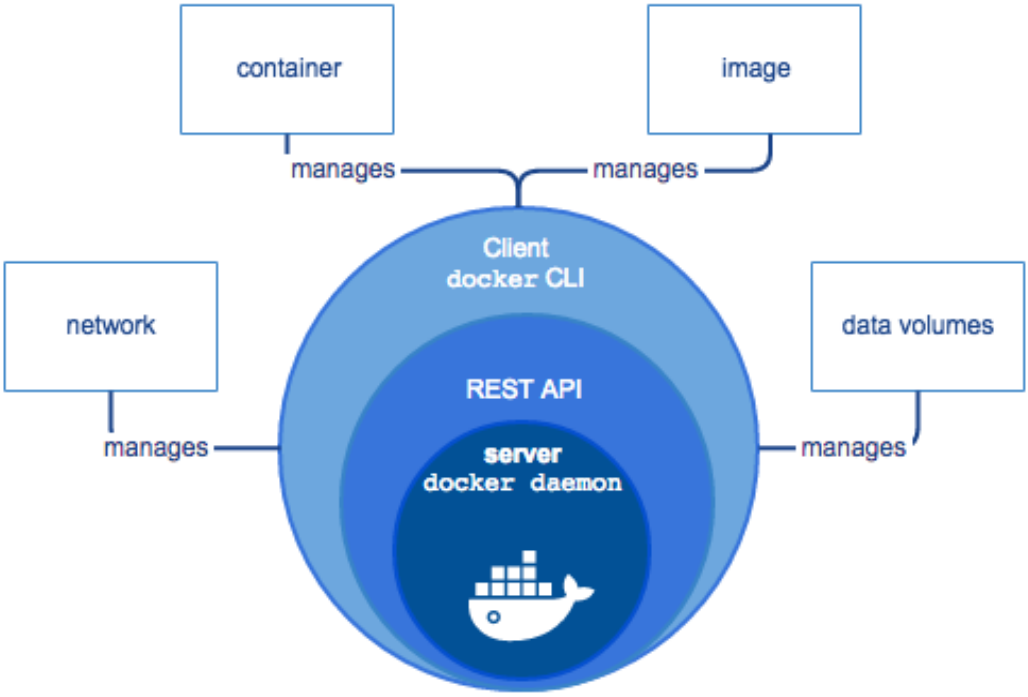


Figure 2. Docker Engine components (Docker Inc. 2020d).

Docker also comes with its own container orchestration tool Docker Swarm, which enables connecting and controlling multiple containers operating either on a single or several computers colloquially called nodes. Newer versions of Docker Desktop for Windows and MacOS also include alternative local development tools for Kubernetes, the current industry standard tool for container orchestration.

Containerd, Docker Engine's original runtime environment has been made available to the public for use by other containerization technologies since 2015. The

aim of this move was to provide a universal industry standard for containers under the Cloud Native Computing Foundation (CNCF) (Docker Inc. 2020e).

3.1.1 Containers

A container is a standardized unit of software that packages up code and all its dependencies together so that it can function identically on any computing platform. (Docker Inc. 2020b). Containers are synonymous with today's microservice development as they by their nature solve many of the fundamental challenges associated with microservices. Because of this, understanding containers is imperative to designing microservice architecture-based applications effectively.

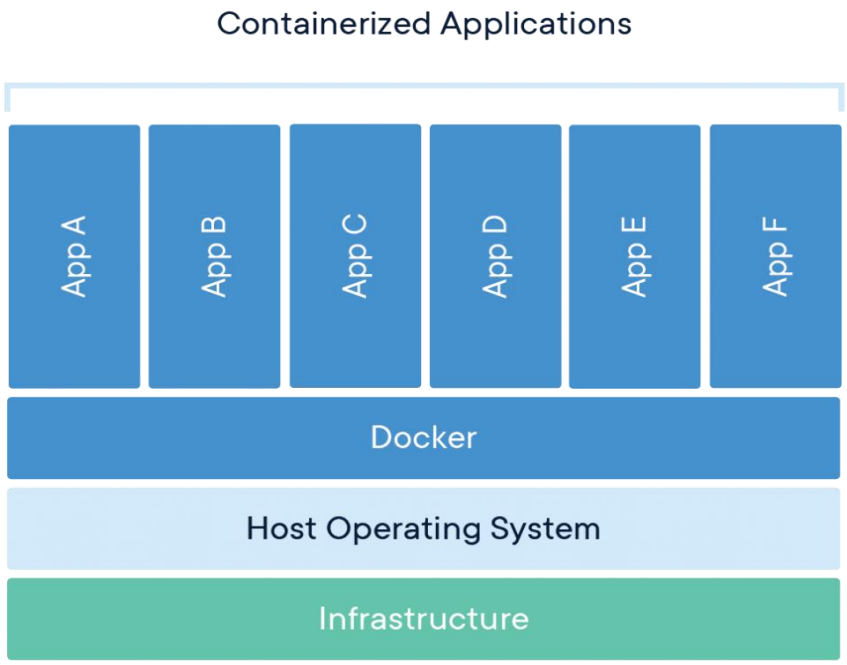


Figure 3. Containerized Application in Docker (Docker Inc. 2020f).

Containers are run on an intermediate operating layer that separates them from the host operating system's runtime environment. This is what enables interchangeable functionality regardless of host operating platform. Every container only includes the bare essentials needed to support its primary running process. This can include a base operating system, network interface, additional programs and other

necessary dependencies. Containers are typically headless, as in capable of working without a graphical interface.

An image is required to start up a container. A single image describes a single type of container but the number of running containers per image is virtually unlimited. Containers are ephemeral, meaning their lifetime is tied to their primary running process. Once that process exits, the container will die, taking all its data down with it.

Containers are often compared to virtual machines by virtue of both utilizing a hypervisor to operate independently from the underlying computing platform of choice. However, there are several key differences between the two that set them apart. Rather than starting up an independent kernel with every instance like virtual machines do, containers simply inherit it from the docker engine. This provides several benefits like faster startup times and better host system resource allocation.

3.1.1 Image

Unlike virtual disk images, Docker images are stored encrypted internally inside Docker Engine's filesystem. Images are built upon intermediate layers that are cached during the build phase with each instruction interpreted as a single layer. If an instruction is completed successfully, the new layer is then appended to the existing layer delta. (Docker Inc. 2020g). The aim of this type of filesystem is to add additional security while enabling faster image build- and transfer times by reducing overall disk usage.

Images do not have names in the conventional sense. Instead they are given unique randomly generated identifier strings and optionally, user defined tags. The purpose of a tag extends beyond giving images human readable identification. A tag is made up of 3 parts: the target registry, the target repository within that registry and finally, image version. Registry tag is the URL where the registry can be reached, followed by a trailing forward slash. The version tag is separated from

the repository tag by a colon. The complete image tag forms an URL with a path pointing to the requested resource, complete with a version argument.

```
services:
    my-docker-app:
        image: localhost:5000/my-docker-app:v1.0
```

Figure 4. Example of a complete image tag using a locally hosted registry.

To build an image the docker build command is run with a build context to an existing Dockerfile. If no context is given, Docker will automatically scan the current working directory for a Dockerfile. The only requirement for defining an image is that it must be based on another existing image also called a base image. The base image is defined using tag in a Dockerfile using the FROM instruction. Docker will first scan the local image filesystem for the specified base image. In the occasion that it is not found, it will then try to reach the tagged registry and download the image instead.

3.1.2 Dockerfile

Dockerfile is a YAML-file that is used to define and build a Docker image. It can be loosely thought of as the source code for building a specific image. Dockerfile contains the instructions and build context that is passed on to Docker Engine when docker build command is executed (Docker, Inc 2020h). The instructions included in any given Dockerfile are separated into steps. Steps are read from top to bottom in chronological order and can include anything from installing a base operating system and various programs, defining environment variables and build arguments to running scripts or adding labels and comments that provide additional information to other developers.

```
FROM ubuntu:18.04
COPY ./app /app
RUN make /app
CMD python /app/app.python
```

Figure 5. Example Dockerfile with 4 build steps.

FROM directive specifies the base image to use in the custom one. COPY is used to copy over some resources from the host machine. RUN executes a command or script during the build process. Conversely, CMD specifies the command to execute when the container is launched. With Docker build, every step is added onto the built image as an additional layer that takes up some extra memory. Because of this it is not unusual to see very long individual steps in Dockerfiles with the aim of minimizing the total number of build layers and reducing final image file size.

3.1.3 Registry

Docker images can be stored remotely inside an image registry. Registry is a stateless server-side application that is used to store and distribute images for development or production purposes. Registries can be run privately or rented from different cloud computing service providers such as Amazon, Google, or Microsoft. Docker also offers a free public registry, Docker Hub over at hub.docker.com.

3.1.4 Volume

Docker volumes offer a mechanism to store persistent data generated by containers or to share it between different containers that are connected to the same network. Volumes are generally mapped to a directory inside the container where the persistent data is stored or generated. The mapped container directory contents are then mirrored inside the volume (Docker Inc. 2020i).

Alternatively, Docker also provides an option to bind mount persistent data directly from a specified directory on the host system. The key difference between volumes and bind mounts is how mapped data is managed. Volumes, like images are stored internally inside Docker Engine's filesystem and cannot therefore be accessed from the host directly. Also, the direction of the copy process is reversed, with the contents of the host mounted directory mirrored inside the container, instead of vice versa.

3.1.5 Network

Docker networks allow connecting several containers together over a virtual network switch. Containers that are attached to a network do not need to be aware of the other containers they are connected to or whether they are running inside docker at all. When a container is attached to a network it is assigned a random IP address within that network. This address is managed by Docker and can be replaced with a user-defined alias for that specific container for container-to-container connections.

Docker has multiple network driver options for different use cases; Bridge, the default network driver is usually used for communications between applications running in standalone containers. Overlay network drivers allow connecting several docker daemons together in orchestrated deployments that can cover multiple nodes. Host network driver removes network isolation from the container and uses the host machine's network directly. Finally, Macvlan network driver allows the assignment of a MAC address to a container, making it appear as a physical machine on the network (Docker Inc. 2020j).

3.1.6 Docker CLI

Commands to the Docker client are issued via a command line interface program simply referred to as docker. This program is responsible for Docker Engine's system configuration, managing connections to local and remote registries as well as managing all resources including containers, images, volumes, and networks.

The CLI accepts a multitude of commands; Build command can be used to build images from a Dockerfile. Push and pull commands can be used to upload or download images from or to a remote registry. Run command is used to start a container from an image. Image, volume, and network commands can be used to manage Docker's internal resources, to list a few of the most commonly used ones.

3.1.7 Docker Compose

Docker Compose is an additional command line interface tool for defining and running multi-container Docker applications. Docker Compose is called from a command line simply as docker-compose (Docker Inc. 2020k). Whereas Docker CLI interprets image build instructions from a Dockerfile, Docker Compose interprets runtime configuration instructions from a YAML-based compose file. Containers in a compose file are defined as services. Furthermore, all defined services and volumes are automatically networked together using a bridge network driver when the compose stack is run.

Originally Docker Compose was intended for automating container development and testing environments and as such was ill-suited for deployment purposes. The popularity of the tool has caused this doctrine to steadily shift however, with more Compose-specific production-oriented features being implemented with new every new released version.

3.2 Kubernetes

Kubernetes is an open-source container orchestration tool for automating deployment and scaling of containerized applications originally developed and released by Google in June 2014. Nowadays the application is actively maintained by the CNCF. Kubernetes is based off Google's internal Borg container deployment system (The Linux Foundation. 2020a). Several key functions of Kubernetes, such as pods, services and labels are directly from Borg and the system as a whole is based on the expertise and experiences of developers that created and maintained the containerized deployment architecture for Borg.

Once deployed, Kubernetes creates a cluster that consists of one or several physical or virtual machines called nodes that perform various computational tasks. Each node runs its own instance of Docker Engine and is joined to the cluster through an automated networking component kube-proxy that interconnects the daemon kubelet processes on each node together.

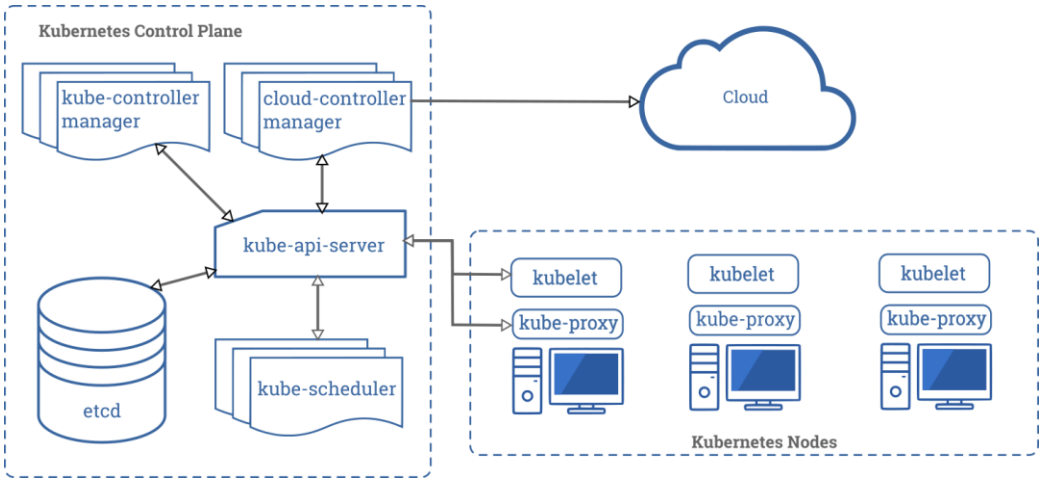


Figure 6. Kubernetes Components Overview (The Linux Foundation. 2020b).

On the surface, the separate kubelet processes are tied together seamlessly as if they were all a single application running on the same host machine. The kubelets can then be accessed through the Kubernetes Control Plane on a separate node designated as the master node that issues commands to the worker nodes in the

cluster through a set of APIs referred to as kube-api-server. Any changes to the state of the cluster is interpreted by the three other processes on the master node; kube-controller-manager, kube-scheduler, and cloud-controller-manager which in unison oversee and manage the worker nodes. The function of the master is to maintain the declared state of the entire cluster. In the event of unforeseen changes to the state e.g. an application exiting due to an error, the master will try to restore the desired state and restart the container by issuing commands to the kubelet on the relevant worker node.

The kubelets themselves act as hosts to a number of containerized processes grouped together in pods, the basic execution unit of Kubernetes. Pods encapsulate containerized applications' storage resources, network identity and runtime configuration (Linux Foundation. 2020c). In practice pods can do anything from running and maintaining containerized applications, overseeing access rights to specific parts of the cluster to acting as networking services between other pods.

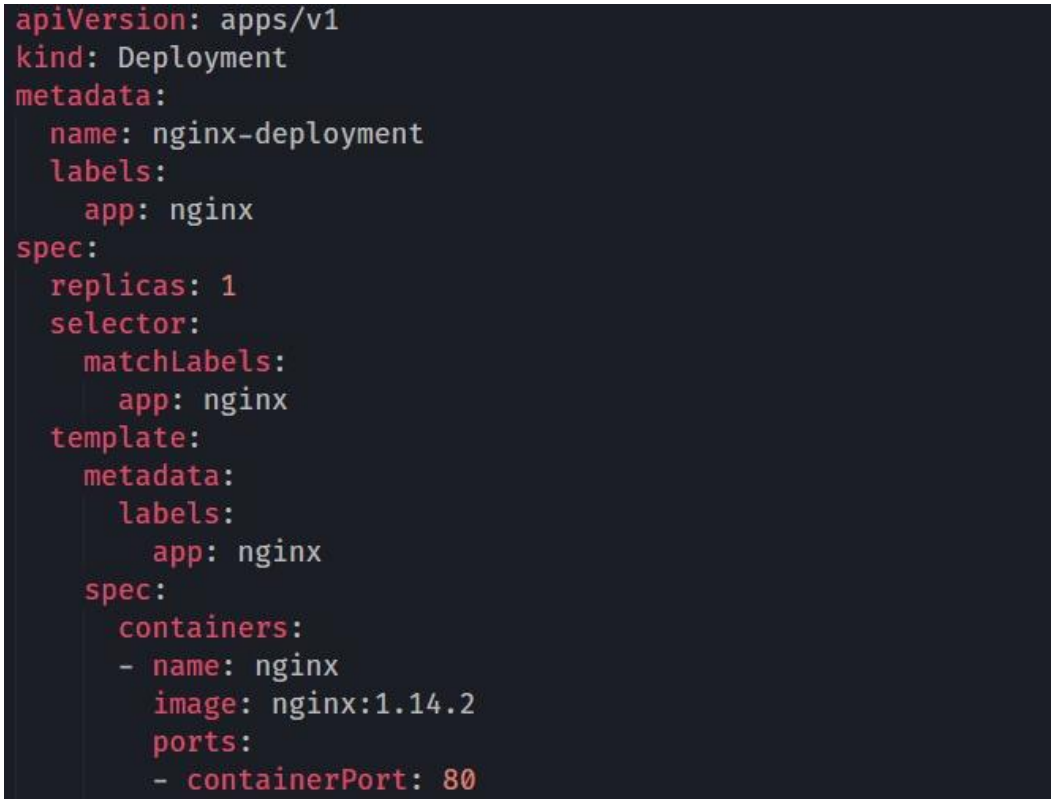
User defined pods can be deployed into Kubernetes through any application that interfaces directly with the Kubernetes Control Plane. In their raw form, pods are defined by YAML documents and fed to the cluster through a command line tool Kubectl. Several web-based user interface applications also exist as an alternative means to do this. Several different types of pod configurations, or objects, exist each with their own purpose and range of configuration options. Typically, these are preferred as means of deploying an application over regular pods, as user-defined pods offer only a limited range of configuration options or must rely on other pods to achieve basic mutability properties required by most types of application deployments.

3.2.1 Deployment

Deployment is an object that provides declarative updates for pods in Kubernetes. A deployment object is defined by setting the desired state of the containers within a pod (The Linux Foundation. 2020d). Deployments are the most common means of deploying production-grade applications because they inherently rein-

force the declarative deployment approach. Declarative object configuration translates to changing the state of the cluster by deploying configuration files instead of directly issuing imperative commands. Because containers are designed to be stateless, this approach leaves behind a reference in the shape of the configuration file itself that can be reused or modified in the future. Taking the declarative approach is strongly advocated for the majority of object definitions amongst both the Kubernetes developers and community.

A deployment YAML-file consists of several nested key-value pairs wherein the multiple configuration options can be defined. Like all objects in Kubernetes, a deployment must conform to the resource assets to use their functionality. This is defined by the apiVersion and kind fields.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 7. Example Kubernetes deployment configuration file.

Uniquely identifying information about the object is described in the metadata-field. All objects are given unique network identifiers or IP addresses within the cluster and the nested name field acts as a hostname within the cluster. Any fields

nested below labels can be used to group and select multiple target objects of matching labels by other objects, typically for networking.

The spec field varies greatly from object to object and defines the specifics of the object's configuration. In the case of a deployment object, the spec states the number of pods or replicas to create in the deployment, their selector labels and a pod template that is used to create them. The value of the image field nested deep inside the template is a tag for the Docker image to use. When the deployment is created, Kubernetes will issue a command to one of its own instances of Docker to start up containers using the specified images.

3.2.2 Service

Services are the networking objects used by Kubernetes to expose and link pods within the cluster. Kubernetes gives pods their own IP addresses and DNS host-names, enabling load-balancing across them (The Linux Foundation. 2020e). The purpose of this method is to remove networking configuration from the deployed applications themselves without abstracting it entirely outside of the cluster architecture.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  clusterIP: 10.0.171.239
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.0.2.127
```

Figure 8. Example Kubernetes service configuration file.

There are multiple kinds of services that can expose applications on specific nodes for development purposes or act as internal network switches inside the cluster, connecting multiple pods together and load-balancing traffic between them.

3.2.3 Ingress

Ingress is a unique type of object that manages external HTTP and HTTPS connections to the cluster. It can be loosely thought of as a network gateway to a Kubernetes cluster. The ingress consists of an ingress controller object and a containerized web server application that serves the actual end-to-end traffic to users. The ingress is typically set up as a reverse proxy and a load-balancer all in one with additional functionality to support various requests coming in and routing it to its designated endpoints. Such functions include but are not limited to giving applications externally reachable URLs, terminating TLS encryption or offer name-based virtual hosting (The Linux Foundation. 2020f).

Kubernetes does not include any implementations of an ingress out of the box, but rather leaves them to 3rd parties, allowing for a wide variety of different solutions for different use cases. Good examples of this are cloud service provider options,

that need to connect to the service provider's server-side load-balancers to map network connectivity between the cluster and complex network interfaces. In such scenarios, the separate nodes managed by the cluster may be physically very far apart from each other or serve traffic for multiple different cloud services under different domains.

As with all other objects, the ingress can be defined and customized in a YAML configuration file that is applied to Kubernetes. The definition options must follow the guidelines set by the 3rd party implementation of ingress that is used. Commonly used implementations include the Nginx-Ingress, based on the popular open-source web server Nginx as well as Traefik, an auto-configuring micro-service-centered load-balancer built specifically for containerized microservice architecture.

3.2.4 Volumes

As mentioned in the Docker section, containers are ephemeral and cannot be used to store persistent data, Kubernetes offers several means to solve this problem. Persistent data is stored inside special objects of several kinds: Volume, persistent volume, persistent volume claim, configmaps and secrets. Regular Kubernetes volumes are similar to those used in Docker. They create a virtual storage space inside a virtualized file system that can be only accessed directly by containers. In Kubernetes, volumes are tied to specific pods, and pods, like containers are also ephemeral, meaning this kind of object is unsuitable for persisting data. Volumes are instead used to share data between separate containers within a pod without intent of storing it permanently (The Linux Foundation. 2020g).

Persistent volumes however, mimic the behavior of Docker bind mounts, specifying a directory on one of the nodes and mapping the volume to a directory inside a container. Persistent volumes may also be bound mounted to remote cloud storage options with extensive coverage of all the biggest cloud service providers' different storage systems. This way data is not only persisted across the entire cluster but outside of it as well. Persistent volume objects may be created imperatively or

declaratively using persistent volume claims. A persistent volume claim is another type of storage option that advertises persistent volumes of different types and sizes available for use by other objects, that can use them by making a claim that the master will then attempt to fulfill.

Configmaps and secrets differ from other types of volumes in that they are not meant to store persistent data required or generated by the containers. Instead they store persistent container configuration in the form of key-value pairs that may declare environmental variables within the containers. Configmaps store non-sensitive data such as application settings, hostnames, or port mappings, while secrets store sensitive data in an encoded form optionally locked behind role-based access control (RBAC) methods. Examples of such data includes but is not limited to login credentials, private keys, or SSL certificates.

3.2.5 Kubectl

Kubectl is the Kubernetes alternative to Docker CLI. It acts as primary developer tool for interfacing with the Kubernetes Control Plane. It can be used to get information about any objects in a cluster using the get and describe commands. Kubectl can also create new objects or delete existing ones with imperative commands create and delete as well as apply new or updated configuration files using the apply command. The full list of commands and their parameters are all listed on the official Kubernetes documentation page (The Linux Foundation. 2020h).

3.2.6 Minikube

For setting up the Kubernetes development cluster, the fastest way to get started is with a program called Minikube. Minikube downloads an image and uses it to provision a virtual machine using a virtualization hypervisor. Alternatively, the cluster VM can also be run inside a Docker container in native Linux environments. The VM created by Minikube acts as a single-node Kubernetes cluster that is capable of serving as a development environment for the majority of typical test deployment purposes (The Linux Foundation. 2020i).

3.2.7 Helm

Helm is a popular 3rd party Kubernetes package manager created by Platform9 that allows packaging of configuration files into installable packages called charts. The purpose of Helm is to make Kubernetes application deployment simple by standardizing the deployment process. The packages can be hosted on public or private Helm repositories that may then be manually installed by others into their own cluster. Helm also provides options to update installed packages or rollback to earlier versions.

A Helm chart contains the metadata of the package, including its name, version info, description, and additional information such as maintainer contact details, links to homepages or documentation etc. Charts also include a templates directory that contains the configuration file templates where all configurable package values are mapped to a values YAML-file. The values are set by the vendor to sensible defaults with the fields commented to allow for end-user customization. This enables streamlined installation of 3rd party applications into any cluster without having to understand the complete application structure and set of configuration options (Parikh 2020).

3.3 Unix Shell

As of April 2020, all 15 of the top Docker container base operating systems are Linux-based distributions (Docker Inc. 2020l). As the entire container technology stack has been built from scratch to replicate the most commonly used web application deployment target architecture, the Unix shell is prevalent throughout all stages of containerized development. Although images for other base operating systems such as Windows Server do exist, the default headless container mode of operation heavily favors Unix-based runtime environments that can utilize the minimal kernel native to the system.

Due to the headless nature of containers, doing virtually anything inside a Linux-based build process requires knowledge of the operating system. This applies to containerized application configuration and debugging as well, as they may differ

greatly from their counterparts on other operating systems. With containers, it is also common to see use cases of altering the container's runtime process using Unix shell scripts.

Although both Docker and Kubernetes are operating system agnostic. Both have their command line interface tools' commands and runtime environments structured based on the Unix environment. Quite often other operating systems require additional configuration steps or workarounds for passing file paths or environmental variables between the host machine and the program in question. It is difficult to deny that Unix is elemental to container development as it is embedded into every step of the process.

3.4 Microsoft Azure

Cloud platforms are at the core of modern web development and microservices are no exception. Microsoft's Azure provides a multitude of cloud-native services related to containers, including hosting private image registries, running cloud-native container instances and a global, scalable, Kubernetes cluster service (Microsoft. 2020a). Azure cloud considerably lowers the bar of entry to deploying world-wide containerized applications due to its high level of service and extensive documentation. Many of these services can be difficult and very time consuming to set up manually and due to their nature would have to be self-hosted and actively maintained for around the clock availability, possibly in multiple geological locations. While taking such an approach is certainly doable and comes with some perks, it is also likely to incur higher costs overall.

Cloud service platforms are often complex entities and centralizing the necessary cloud services to a single provider is key in being able to manage all the required tools under a single portal. This can provide other benefits as well, such as ease of management of various assets and access rights under separate services. This thesis covers such a scenario by integrating an image build process directly to an Azure container registry with Azure DevOps build pipeline running in the cloud.

3.4.1 Azure Container Registry

Azure Container Registry is a private, web managed image registry cloud service based on the newest Docker registry version 2.0. The registry is configurable within the Azure cloud subscription with integrated features, such as a visual web dashboard, secure RBAC authentication, geo-replication, automated tasks and direct access to private Azure artifacts and Azure DevOps Pipelines (Microsoft. 2020b).

3.4.2 Azure DevOps Pipelines

Azure Pipelines is an automated building and testing cloud service that can checkout git repositories and Azure hosted artifacts into a VM provisioned directly to the cloud and run a series of pre-configured tasks on them. If the repositories and assets exist on the Azure DevOps git, authentication procedures will be automated. The pipelines may also be executed automatically using push or pull request triggers (Microsoft. 2020c). Typical use cases of pipelines in container development would be to automate Docker image build procedures and either pushing the images to an image registry or directly deploying them to a Kubernetes cluster.

3.4.3 Azure CLI

Azure CLI is the primary developer tool for interfacing with Microsoft Azure Cloud services through the command line. Azure CLI is used to create and manage subscriptions, resources, and access rights with an emphasis on automation. The tool is available for Windows, macOS and Linux environments and can also be run inside Docker (Microsoft. 2020d).

4 DEVELOPMENT PROCESS

4.1 Installing Docker and Kubernetes

4.1.1 Windows and macOS

Docker installation varies depending on the operating system. For Windows and macOS environments, the preferred installation method is the official pre-configured installation of Docker Desktop. For Docker Desktop installation certain system requirements must be met. The program is only supported on newer 64-bit OS versions such as Windows 10 and macOS 10.13 or newer. Virtualization features must also be enabled in the BIOS and a virtualization hypervisor such as Hyper-V (Windows) or HyperKit (macOS) must be enabled. Docker Desktop comes with both Compose and Kubernetes development tools included, latter of which can be manually enabled from the settings menu.

For systems that fail to meet these requirements, a version of Docker Toolbox may be used instead. Docker Toolbox uses the popular open-source hypervisor Oracle VM VirtualBox, which must be installed separately. The installation process for Docker Desktop and Docker Toolbox options is rather simple with clear and straightforward instructions available online.

4.1.2 Linux

For Linux operating systems, Docker Engine should be installed instead. The installation process can vary based on the target system distribution. Fortunately, there are multiple installation options including using package managers, running an installation script, downloading prebuilt binaries, or compiling the program directly from source code. In this project, the installation was done on the popular Debian-based Linux distribution Ubuntu Bionic 18.04 LTS, using the Apt package manager. The Ubuntu installation via Apt is covered separately in the Docker documentation.

It should be noted that the Optional post-installation steps section on the documentation page includes steps to create a user group called docker, which enables use of the docker program without invoking super user access every time commands are issues to the program. Other useful tips and tricks can be found on this page and reading through it is highly recommended.

Docker Compose must be installed separately on Linux systems. If the Apt repository is added to the source lists, the package can be installed by simply adding the package name docker-compose to the end of the second install command, as demonstrated in the install commands table below.

Table 1. Docker Engine install commands.

Command	**Description**
`sudo apt-get update`	Update the Apt package index.

`sudo apt-get install \`
`  apt-transport-https \`
`  ca-certificates \`
`  curl \`
`  gnupg-agent \`
`  software-properties-common`
	Install required software dependencies.

`curl -fsSL`
`https://download.docker.com/linux/ubuntu/gpg`
`| sudo apt-key add -`
	Add Docker's official GPG key.

`sudo add-apt-repository \`
`  "deb [arch=amd64]`
`https://download.docker.com/linux/ubuntu \`
`  $(lsb_release -cs) stable"`
	Add Docker stable repository to the sources list.

`sudo apt-get update`	Update the Apt package index.
`sudo apt-get install docker-ce docker-ce-cli` `containerd.io docker-compose`	Install Docker and Docker Compose.

Kubernetes development tools installation process is different depending where and how the development cluster is set up. At the very least Kubectl is required

for issuing commands to the cluster. In this project a development cluster environment is run inside a VM provisioned by Minikube.

Both Kubectl and Minikube will be downloaded and installed as prebuilt binaries. The definitive up-to-date installation instructions for Kubectl and Minikube can be found on the official Kubernetes documentation web site.

Table 2. Kubectl install commands.

Command	**Description**
`curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-re-lease/release/stable.txt`/bin/linux/amd64/kubectl`	Download the latest version of Kubectl.
`chmod +x kubectl`	Make the binary executable.
`sudo install kubectl /usr/local/bin/`	Move the binary into PATH.

Table 3. Minikube install commands.

Command	**Description**
`curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64`	Download the latest version of Minikube.
`chmod +x minikube`	Make the binary executable.
`sudo install minikube /usr/local/bin/`	Move the binary into PATH.

In addition to Kubectl and Minikube, the Kubernetes package manager Helm is also installed in order to package the final application into an easily installable

format. The installation is done using an automated installation script that can be found on the Helm web site.

Table 4. Helm install commands.

Command	**Description**
`curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3`	Download the Helm install script.
`chmod +x get_helm.sh`	Make the script executable.
`bash get_helm.sh`	Run the script using Bash.

4.2 Getting Started

Docker Hub provides all the necessary information to get started on creating a custom image and should usually the first stop for information. Other resources used for containerizing any application are usually its official documentation complimented with a myriad of freely available online guides and tutorials that guide you through the initial process step-by-step.

When creating images, the best practice is to keep everything to its bare minimum requirements and think modularly. An image should contain no more or less than what it needs to accomplish its intended purpose, but it should also be able to be quickly adapted from one system to another. All hard-coded values that may need to be configured for a specific use case in the future as well as all connection values including hostnames, ports and login credentials should be bound to environment variables. As a rule of thumb, all references to either localhost or 127.0.0.1 can and will cause problems in testing. This is because Docker containers do not operate on the host network, and each have their own network configuration.

Data persistence also needs to be taken into consideration. If not persisted, all data generated by the application is lost when the container crashes or is stopped. That

includes cache, file uploads, error logs, and even database storage. In some cases, even if the data does not need to be persistent it might need to be accessed by multiple containers.

Things to take into consideration when creating a custom image:

- What is the intended purpose of the image?
- What programs are needed to fulfill that purpose?
- What are the required dependencies for those programs?
- What base image should be used?
- What external connections need to be made?
- What data needs to be persisted outside of the container?
- What needs to be configured or may need to change in the future?

4.3 Nginx Gateway

All web applications require a web server that listens to incoming requests and returns a response to the client. In microservice architecture, that server needs to forward the requests on behalf of the client to one or more upstream servers and return their response to the client without revealing its true origin. This type of server is known as a reverse proxy. This server ultimately serves the entire web application's content to the outside world, acting as a gateway; the single point of access to the services connected in the application's internal network. One of the most popular programs to act as this type of service is the open-source web server Nginx.

As the gateway into the system, this service is responsible for everything that is expected of a production-grade web server. It needs to expose ports, contain the SSL certificates, redirect plain HTTP traffic to HTTPS, serve the UI, set the proper response headers, route requests upstream and set other various connection settings and request permissions.

All this can all be configured in the application settings files. The easiest way to alter them is to run a Nginx container and manually copy the default settings di-

rectory to the host machine for editing. The edited files can then be copied over to a custom image that is based on the official one, overwriting the default files. This method allows for the rest of the preconfigured official Nginx image to stay intact.

Table 5. Copying files from containers.

Command	**Description**
`docker run -d nginx:1.17.8-alpine`	Run a Nginx container in the background.
`docker ps`	Print the id of all running containers.
`docker cp <container-id>:/etc/nginx ./`	Copy configuration files to host.
`docker rm <container-id> -f`	Kill and remove a container.

As shown in the first command, the Alpine Linux version of Nginx is used as the base image option for the gateway service. Generally Alpine provides the best base for building images due to its tiny file size of just over 5 MB and its included package manager APK's minimal, recursive handling of software dependencies. Alpine also boasts higher built-in security features as it has been designed from the ground up to be used in IoT devices and containerized workloads. Other distributions should only ever be used if the required software dependencies absolutely depend on it.

Once the default configuration files have been edited, they can be simply copied back to their original directory on the custom image using the Dockerfile COPY directive. Only the files that were changed need to be kept. The other files can be safely deleted as their default versions are already found on the base image.

Table 6. Building and running custom images.

Command	**Description**
`docker build -t gateway:dev .`	Build and tag image from Dockerfile.
`docker run -p 80:80 gateway:dev`	Run the custom image and publish port 80.

In the main nginx.conf file, inside the http block other configuration files are included to separate server configuration settings. These four .conf files make up the manually edited Nginx configuration files.

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
    charset utf-8;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;

    keepalive_timeout  65;

    include /etc/nginx/conf.d/gateway.conf;
    include /etc/nginx/conf.d/upstream.conf;
    include /etc/nginx/conf.d/proxy_headers.conf;
}
```

Figure 9. The main Nginx configuration file.

In the file upstream.conf, the individual services and their hostnames and exposed ports are defined. The hostnames will be later mapped automatically to the randomly assigned individual container IP addresses in Docker Compose.

```
# Apache
upstream apache {
    server apache:80;
}

# Go
upstream go {
    server go:8080;
}

# MariaDB
upstream mariadb {
    server mariadb:3306;
}
```

Figure 10. Nginx upstream configuration file.

Then in the file gateway.conf, which contains the virtual HTTP server, inside the server block, incoming requests to specific endpoints can be routed upstream as necessary. The response headers defined in separate files like proxy_headers.conf can also be included here for customizing headers to specific upstream services.

```
    # Backend - Go
    location /index.php/auth/ {
        include /etc/nginx/conf.d/proxy_headers.conf;
        proxy_pass  http://go;
    }

    # Backend - PHP
    location /index.php/auth {
        include /etc/nginx/conf.d/proxy_headers.conf;
        proxy_pass  http://apache;
    }
}
```

Figure 11. Nginx HTTP server endpoint routing.

These configuration files are copied onto the base image during the build phase overwriting the defaults. The built Angular 6 UI assets that contain all of the backend calls are also be copied over to this image to be served by Nginx.

```
FROM nginx:1.17.8-alpine

# Copy application files
COPY ./ui /var/www/ui

# Copy config files
COPY ./config /etc/nginx

# Run nginx as the main active process
ENTRYPOINT  [ /usr/sbin/nginx -g "daemon off;" ]
WORKDIR     /etc/nginx
EXPOSE      80 443
```

Figure 12. Gateway Dockerfile.

The ENTRYPOINT directive tells Docker which process to run when the container is started. The WORKDIR and EXPOSE directives each have their own special uses, but in this scenario, they were included for commentative purposes only.

4.4 Apache2/PHP

This service contains the employer's current PHP backend git repository and serves the application using the open-source Apache2 web server. Since PHP is an interpreted programming language, all application source code must be copied over to the image during build and interpreted using a compatible installation of PHP. This service needs to connect to the database for the application to function so the connection configuration needs to be altered. As the Apache2 web server is located behind the Nginx gateway, there is no need for providing SSL certificates or setting response headers. As such the server configuration can be kept to a minimum.

With the employer's current plans to gradually replace this backend entirely using Go, attempting to optimize this service would be meaningless. Additionally, there are some PHP version constraints in place that further limit the available base image options. The simplest way to implement this service is to simply replicate the current server initialization steps as closely as possible inside a Docker environment. That means using the ubuntu:18.04 base image instead of the available offi-

cial PHP or Apache2 image variants and then attempting to manually incorporate the other or trying to implement them in separate containers.

```
FROM ubuntu:18.04

# Set OS timezone
ENV TZ=Europe/Helsinki
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

# Update & upgrade packages
RUN apt-get update -y && \
    apt-get install software-properties-common -y --no-install-recommends

# Add repositories
RUN add-apt-repository ppa:ondrej/php -y && apt-get update

# Install Apache2 & PHP 7.0
RUN apt-get install -y --no-install-recommends \
    apache2 \
    php7.0

# Install PHP 7.0 extensions & clear apt lists
RUN apt-get install -y --no-install-recommends \
    libapache2-mod-php7.0 \
    php7.0-mysql \
    php7.0-common \
    php7.0-curl \
    php7.0-imagick \
    php7.0-mcrypt \
    php7.0-mysqli \
    php7.0-json \
    php7.0-intl \
    php7.0-memcache \
    php7.0-mbstring \
    php7.0-fpm \
    php7.0-gd \
    && apt-get clean && rm -rf /var/lib/apt/lists/*

# Copy application files
COPY ./appdata /var/www/appdata

# Copy configuration files
COPY ./config /etc/apache2

# Copy and xmod scripts
COPY ./scripts /scripts
RUN chmod +x /scripts/*

# Run apachectl as active process
ENTRYPOINT  [ "apachectl" ]
CMD         [ "-D", "FOREGROUND" ]
WORKDIR     /var/www/appdata
EXPOSE      80
```

Figure 13. Custom Dockerfile from Ubuntu base image.

4.5 Go Microservices

As the employer's current backend revision is still underway, only a single service written in Go can be made during the project time constraints. The Go service

contains an authentication service, that separates session management from the PHP backend under a different session cookie. The original cookie served by the old backend is still necessary however, at least until the new one is finished. Because of this the Go service needs to make an additional request to the Apache2/PHP service during authentication requests and return both session cookies to the client. These kinds of rogue requests between two independent microservices are not usually appropriate within microservice architecture design but in this case an exception had to be made.

The Go service is therefore intended as more of a reference for the future and perhaps even more than that, a prime example of the superior nature of Go's design when it comes to developing containerized microservices. Unlike PHP, Go is a compiled language that is inherently designed for developing asynchronous web applications with lightweight built-in web server functionality. Go also includes its own package manager, which allows for easy dependency tracking and automated builds.

In production, Go source code is compiled to an executable binary native for the target OS, which reduces of the risk of unwanted 3rd parties gaining access to the files while also cutting out the need of installing any additional runtime environment software. In practice this results to greatly reduced overall image sizes which correlates directly to faster image build, pull and container startup times. On top of that, Go's concurrency model enables it to rival the performance of many traditionally lower level programming languages such as C++ under multithreaded workloads.

As this image is not intended for production purposes, it raised an opportunity to demonstrate one of Docker's more advanced image build options known as multistage builds. Instead of compiling the binary externally and then copying it to the image, the Go source code is copied to a custom image based on the official golang:1.13.8-alpine3.11 image designated as builder. The builder is then given two RUN directives to first download all dependencies listed in the included Go package manifest and compile the source code into a binary inside Docker.

The alpine:3.11 base image is then declared using FROM directive followed by a COPY directive to copy just the built binary from the first stage. The image of the first stage can then be discarded as a build artifact resulting in a production-ready containerized Go microservice totaling less than 20 MB in file size.

```
FROM golang:1.13.8-alpine3.11 AS builder

# Copy source files
COPY ./src /go/src

# Install dependencies
RUN cd /go/src && \
    go get -v all

# Build binary
RUN cd /go/src/cmd/go-api && \
    go build -o /go/bin/go-api


FROM alpine:3.11

# Copy binary from builder
COPY --from=builder /go/bin/go-api /app/go-api

ENTRYPOINT  [ "/app/go-api" ]
WORKDIR     /app
EXPOSE      8080
```

Figure 14. Multi-stage build in Dockerfile.

4.6 Mariadb Database

The official database images hosted on Docker Hub normally come with additional Docker functionality and require the least amount of manual setup. The official Mariadb image for example includes a database initialization script, that will be automatically run on container startup. The script will look for any .sql and .sh scripts under the directory /docker-entrypoint-initdb.d and executes them before the database server is brought online. By simply bind mounting a SQL database dump to that directory when the container is started will automatically trigger the initialization process producing a fully functional development database.

Also, a series of environmental variables are already declared ahead of time that the initialization script will use when creating databases or login credentials with adequate permissions to use the database. This service requires no Dockerfile.

4.7 Docker Compose

With the individual images built, it is now possible to test them together. One way to go about this is to create a Docker network, manually run each container with correct port and volume parameters mapped and attach them to the network one-by-one. This is where Docker Compose comes in. With this approach it is possible to define each service with its port mappings and volumes inside a compose file into a compose container stack. This file can then be used to execute as a single process that creates a bridge network, parses the individual run, and attach commands in their entirety and brings the entire stack online at once.

```yaml
version: '3.3'
services:
    apache:
        image: apache:dev
        build:
            context: ./apache
        container_name: apache
    go:
        image: go:dev
        build:
            context: ./go
        container_name: go
    mariadb:
        image: mariadb:dev
        build:
            context: ./mariadb
        container_name: mariadb
        volumes:
            - ./mnt/sql/dump.sql:/docker-entrypoint-initdb.d/dump.sql:ro
            - ./mnt/data:/var/lib/mysql:rw
        ports:
            - 3306:3306
    gateway:
        image: gateway:dev
        build:
            context: ./gateway
        container_name: gateway
        ports:
            - 80:80
            - 443:443
```

Figure 15. Docker Compose file.

Each service name is used as the hostname for the container connected to the bridge network. Docker build context must also be explicitly specified if the compose file is to be used for building the listed service images. Container names can also be set here, enabling easier developer access to the containers using their specified names instead of randomly generated identifier strings. The docker-compose binary also accepts common parameters from docker such as -d to run the stack as a daemonized background process.

Table 7. Docker Compose commands.

Command	**Description**
`docker-compose up`	Run the Docker Compose stack.
`docker-compose down`	Stop and remove Docker Compose stack.
`docker-compose build`	Build all images in a Docker Compose stack.

4.8 Extending Container Functionality

With the application having cleared the initial testing phase, it needs to be evaluated in something resembling a production environment. After all, one of the core tenets of Docker is to bridge the gap between development and production environments and reduce the manual labor required to shift software products between them. For this it the application needs to be served over HTTPS.

The Gateway service publishes two ports: 80 for HTTP and 443 for HTTPS connections originating from any client. Those ports are mapped by Docker to the host machine's network and can be accessed locally through a web browser by navigating to http://localhost and https://localhost. As HTTPS requires valid SSL certificates accessible to the Nginx server, trying to serve encrypted traffic would cause the application to crash so long as they are missing.

Certificates trusted by common web browsers are cryptographically signed by trusted 3rd parties known as a certificate authority (CA). The next goal of this project is to automate the requesting and periodic renewal of such certificates under a valid domain name from the CA Let's Encrypt using Certbot, a tool designed for that very purpose.

Certbot needs to share the same filesystem with the Nginx web server as that server will be used to serve a plain HTTP DNS challenge to Let's Encrypt that proves that whoever is requesting certificates is the true owner of the domain the certificates are requested for. The same process that triggers the Certbot request process also requires control access to the Nginx running in Docker in order to restart the server once the certificates have been granted.

This workflow poses some critical problems for Docker. Firstly, there are no certificates in place when the container is started with SSL enabled. And second, Nginx, the active process running inside a container needs to be restarted at some point. In either case, in its current state the container would crash immediately. There are a few workarounds available and all of them require writing Unix shell scripts.

4.8.1 Container Startup Scripts

Instead of using the ENTRYPOINT directive in the Dockerfile to give control of the container to Nginx, it is possible to pass it to a shell script instead. This is a common practice in containers that require some amount of runtime setup that cannot be implemented during the build phase. In this scenario that script would need to issue commands to both Nginx and Certbot, which requires placing them inside the same container.

While this works in practice, there are still some minor issues taking this approach. Ideally, containerized microservices aim at the total separation of processes into individual containers. If anything, situations such as these underline that Docker is intended first and foremost to be used as a development tool and not for production.

```
#!/bin/sh

# Request certificates
certbot certonly --standalone -d some.domain.fi --email some@email.fi -n --agree-tos --expand

# Start cron
/usr/sbin/crond -f -d 8 &

# Start nginx
/usr/sbin/nginx -g "daemon off;"
```

Figure 16. Custom entrypoint script.

Another script certbot-renew is included as well to automate the certificate renewal requests. The execution of this script is handled by the crond process that automates periodic tasks at certain time intervals on Unix systems.

```
FROM nginx:1.17.8-alpine

# Install Certbot & create DNS challenge dir
RUN apk add --no-cache certbot && \
    mkdir -p /var/www/certbot

# Copy application files
COPY ./ui /var/www/ui

# Copy config files
COPY ./config /etc/nginx

# Copy scripts
COPY ./scripts/entrypoint.sh /entrypoint.sh
COPY ./scripts/certbot-renew.sh /etc/periodic/daily/certbot-renew

# xmod scripts
RUN chmod +x /entrypoint.sh && \
    chmod +x /etc/periodic/daily/certbot-renew

# Execute script entrypoint.sh
ENTRYPOINT  [ "/entrypoint.sh" ]
WORKDIR     /etc/nginx
EXPOSE      80 443
```

Figure 17. Improved Gateway Dockerfile.

With the scripts in place, the gateway Dockerfile needs to be edited. Certbot is installed and given its own directory under the server root directory where it can send the request and respond to the returned Let’s Encrypt DNS challenge.

Finally, the Nginx virtual server configuration is edited to use the newly signed SSL certificates and set some additional SSL security policies that are recommended by Let's Encrypt. Once the container comes online, the entrypoint script will request the certificates and trigger the renewal cron script. After that, the Nginx will be started with the new configuration and the application can be served over HTTPS.

Let's Encrypt certificates are free, but there is a weekly limit to how many can be requested. Instead of storing the certificates in the container, it is usually a good idea to keep them either inside a volume or bind mount them to the host machine for safekeeping.

```
# HTTPS Server: https://some.domain.com:443/
server {
    listen       443 ssl;
    server_name  some.domain.com;

    root         "/var/www";
    index        index.php index.html index.htm;

    error_log    /var/log/nginx/error.log;
    access_log   /var/log/nginx/access.log;

    ssl_certificate       /etc/letsencrypt/live/some.domain.com/fullchain.pem;
    ssl_certificate_key   /etc/letsencrypt/live/some.domain.com/privkey.pem;

    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers  on;

     # Backend - Go
    location /index.php/auth/ {
        include /etc/nginx/conf.d/proxy_headers.conf;
        proxy_pass  http://go;
    }

    # Backend - PHP
    location /index.php/auth {
        include /etc/nginx/conf.d/proxy_headers.conf;
        proxy_pass  http://apache;
    }
}
```

Figure 18. Nginx HTTPS server configuration.

4.8.2 Environmental Variables

The service stack is not yet production ready. It can serve HTTPS, but it can only ever do so on a single domain using certificates that are signed specifically for it. If the domain were to change at any point in time, many of the hardcoded values would need to be edited manually. Those hardcoded values can be replaced with references to environment variables that have their values injected when the container is started. This will allow the contents and functionality of images to stay the same regardless of the host environment.

Before starting any containers, Docker Compose by default looks for a specific file in the same directory along the Dockerfile named .env. These files may contain declarations of environment variables that are then exported to the container's

runtime environment where they can be used globally. Env files were implemented alongside every single Dockerfile used in this project covering all values that could change at any point in the future. For reference, the env file for the Gateway service will be covered here.

```
ENVSUBST_NGINX_DOMAIN=some.domain.com
ENVSUBST_HTTP_PORT=80
ENVSUBST_HTTPS_PORT=443
ENVSUBST_APACHE_HOST=apache
ENVSUBST_APACHE_PORT=80
ENVSUBST_GO_HOST=go
ENVSUBST_GO_PORT=8080
ENVSUBST_MARIADB_HOST=mariadb
ENVSUBST_MARIADB_PORT=3306
```

Figure 19. Gateway env file.

While shell scripts can reference these environment variables by name and programming languages can export their values using functions, Nginx does not support using environmental values by default. While it is possible to extend Nginx with this functionality by using customized 3rd party images, a simpler and more lightweight solutions exist. One such solution is to use Envsubst, a Unix shell templating engine that finds references to these variables in the configuration files directory and replaces each reference with its given value from the container runtime environment.

The Envsubst logic is implemented in a separate script which is called by the entrypoint script. To avoid replacing Nginx's own variables, the substituted variables must be specified when the script is called. This is accomplished by using a uniform naming scheme for the variables. The names of all variables that include ENVSUBST are stored as a string in a temporary variable. The variable is then used as an argument for Envsubst to only substitute the specified references in files with a .tpl file extension that can be found under the directory path of /etc/nginx/tpl.

```
#!/bin/sh

tpldir=/etc/nginx/tpl
vars=$(awk 'BEGIN{for(env in ENVIRON) print "$"env}' | grep 'ENVSUBST')

for tpl in $(find $tpldir -name "*.tpl"); do

    conf=$(sed 's/\/tpl//; s/.tpl//' <<< $tpl);

    envsubst "$vars" < $tpl > $conf;

done

# Delete templates
rm -rf $tpldir
```

Figure 20. Envsubst script.

All custom Nginx configuration files can then be templated in the following manner, replacing hardcoded values with references to the variables declared in the env file. Because the environment variables are injected into the container during runtime, this also enables changing configuration options that have their values mapped to environment variables without having to rebuild the image every time.

```
# HTTPS Server: https://some.domain.com:443/
server {
    listen          ${ENVSUBST_HTTPS_PORT} ssl;
    server_name     ${ENVSUBST_NGINX_DOMAIN};
```

Figure 21. Envsubst template.

4.8.3 Dockerignore

Since one of the service images needs to be built with a nested Git repository inside of it, a file called .dockerignore should be included alongside the Dockerfile. Dockerignore files are read automatically by Docker during image build phase and all files and directories listed inside the file are excluded from the build context instead of being copied to the image. This is perhaps a minor but incredibly important step when working with Docker.

All of Docker's source file types can be safely listed inside this file in case the build context contains nested directories of other Docker images. The files located

in the project root directory are not usually copied to the image for any reason and are unaffected.

```
# Exclude from Docker build context
**/Dockerfile
**/*.dockerfile
**/*.env
**/docker-compose.yml

**/.git*
**/.git/
**/*.md

azure-pipelines.yml
**/docs/
```

Figure 22. Dockerignore file.

4.9 Azure Container Registry

The Azure Container Registry offers a centralized location in the Azure cloud to store images for distribution and is necessary for accessing them from a container orchestrator later. Creating a private registry on Azure requires a Microsoft account with an active Azure subscription. The registry can be created through the Azure web portal after which it can be accessed by a few different means.

The web dashboard for the registry requires administrative access. Thankfully, the registry is also available from the Azure CLI to other user accounts that have been granted contributor rights to the resource. Some features remain locked to contributors but Azure CLI still provides the most common functions such as listing repositories and inspecting their contents, as well as pulling and pushing images.

The registry can also be configured with specific access rights for service principal accounts, which are meant to be used by automated processes such as container orchestrators or build pipelines. The service principals do not require Microsoft

accounts but rather authenticate with randomly generated login credentials and optionally, public keys signed by specific hosts as an additional security measure.

Service principal access is fully compatible with Docker's authentication model and can therefore be used to grant image pull access to automated processes without requiring an installation of Azure CLI. The Azure Container Registry and the service principal account used in this project were created by the employer.

4.10 Azure Pipelines

Azure Pipelines was used to set up an automated process to checkout select git code repositories, use them to build images in the cloud and then upload them directly to the private registry. The image build instructions are defined inside Dockerfiles found in every repository and wasn't changed in any manner from a local build process.

The instructions for the pipeline are saved to a file called azure-pipelines.yaml upon creating a new pipeline. The file is automatically added to the git repository acting as the trigger. The entire pipeline can be configured online through the Azure DevOps portal with ready code snippets for general tasks, that can be customized to great extent with the help of the official documentation.

4.11 From Docker to Kubernetes

At this point of the project, progress had to come to a halt as the employer's new backend was still under development. There was another opportunity available, however. A request had come to the company for a containerized prototype of the current web application with integration into an existing Kubernetes production cluster. With some minor changes, the current project could be transformed into a serviceable product offering.

In a Kubernetes environment, an ingress takes the place of the gateway service, handling all its responsibilities and more. As for the Go service, it was more of a temporary solution intended for future reference from the start. Furthermore, the

customer would be providing their own database implementation, so the only component that would have to be delivered was the Apache service, configured for Kubernetes and packaged into an easily installable Helm chart.

The Helm chart should only include the necessary object definitions to run this single application in Kubernetes. That involves the deployment that runs the application container, a cluster IP service for networking with an ingress, a persistent volume claim for storing user file uploads and two secrets; One for storing database login credentials and another for authenticating with the company's private registry to download the image.

4.11.1 Creating a Development Cluster

To run a local development cluster a Minikube VM is created. Kubernetes interface configuration will be generated automatically to allow Kubectl to issue commands and apply configuration files to the cluster. The cluster can be reached through the browser by navigating to the IP address of the node provisioned by Minikube. appending any ports that the cluster is set to expose for external access. A visual overview of the cluster is also available through the Minikube dashboard.

Table 8. Minikube commands.

Command	**Description**
`minikube start --driver=docker`	Run a local Kubernetes cluster inside Docker.
`minikube addons enable ingress`	Enable the preinstalled ingress compatibility addon.
`minikube dashboard`	Open the web dashboard for an overview of the cluster.
`minikube ip`	Get the IP address of the node provisioned by Minikube.

4.11.2 Declarative Object Configuration

For local testing purposes, an ingress, a database deployment, and a database cluster IP service are necessary and should be created first. The ingress used here is the popular open-source Nginx-Ingress, the documentation and configuration options of which can be found on its Github page. Nginx-Ingress comes preconfigured for development and the only customization required for now is to connect its backend path endpoints to cluster IP services that need to be reachable directly from a web browser. Like with Docker Compose, the service names act as hostnames inside the cluster's internal network.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: nginx-ingress
    namespace: default
    annotations:
        kubernetes.io/ingress.class: nginx
        nginx.ingress.kubernetes.io/rewrite-target: /
spec:
    rules:
     - http:
        paths:
         - path: /
           backend:
            serviceName: app-clusterip-service
            servicePort: 80
```

Figure 23. Nginx-Ingress configuration file.

A database deployment and service can then be created. This time the SQL scripts that initialize the database cannot be simply bind mounted to the image, because the Docker Engine that starts the container exists inside a VM. The database initialization scripts will have to be copied to a custom image by creating a Dockerfile and using the COPY directive. That image can then be passed over to Kubernetes where it will be run inside the database deployment pod, which will be connected to its cluster IP service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: mariadb-deployment
    namespace: default
spec:
    replicas: 1
    selector:
        matchLabels:
            app: mariadb
    template:
        metadata:
            labels:
                app: mariadb
        spec:
            containers:
             - name: mariadb
               image: mariadb:k8s
               ports:
                - containerPort: 3306
               env:
                - name: MYSQL_DATABASE
                  value: somedb
                - name: MYSQL_ROOT_PASSWORD
                  value: somepw
```

Figure 24. Mariadb deployment configuration file.

```yaml
apiVersion: v1
kind: Service
metadata:
    name: database-clusterip-service
    namespace: default
spec:
    type: ClusterIP
    selector:
        app: mariadb
    ports:
     - port: 3306
       targetPort: 3306
```

Figure 25. Mariadb cluster IP service configuration file.

Kubernetes expects all images it uses to be already built. All the developer can do is provide the name of the image. In the case it does not exist on any of the nodes, Kubernetes will try to pull it to its Docker daemon from a remote registry. While

using a remote registry would certainly work, it requires multiple steps to overcome what is essentially a very low hurdle. Information would have to travel potentially hundreds of kilometers over the internet to be placed only some nanometers apart from its original location.

A much better solution is to take advantage of Docker's client-server model design. Docker client by default connects to the local Docker daemon, which is essentially a web server. The connection values are set by a collection of environmental variables which can be redefined. This method will allow building an image using build context from the host machine while saving the image to a remote daemon's filesystem. On Linux this can accomplished with a single command.

Table 9. Minikube Docker and Kubectl commands.

Command	**Description**
`eval $(minikube -p minikube docker-env)`	Export Minikube's Docker environment variables to host shell.
`docker build -t mariadb:k8s .`	Build the image from the host machine to Minikube.
`kubectl apply -f k8s/`	Apply a file or directory of multiple files to Kubernetes.
`kubectl get all`	Print all objects managed by the Kubernetes cluster.

Once the image is on the correct Docker daemon, all configuration files can be applied to the cluster by specifying a path to the files using Kubectl and the database deployment should come alive, although it is still unreachable for now.

The application deployment image is stored on the private Azure registry. Rather than building it manually to Minikube, it is intended to be pulled by Kubernetes.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: app-pvc
    namespace: default
spec:
    accessModes:
     - ReadWriteOnce
    resources:
        requests:
            storage: 1Gi
```

Figure 26. Persistent volume claim configuration file.

The complete deployment file is quite long. Only the relevant parts are displayed.

```yaml
spec:
    volumes:
      - name: app-storage
        persistentVolumeClaim:
          claimName: app-pvc
    imagePullSecrets:
      - name: registry-secret
    containers:
     - name: app
       image: someregistry.azurecr.io/app:1.0
       volumeMounts:
        - name: app-storage
          mountPath: /var/www/app/storage
       env:
        - name: MYSQL_HOST
          value: database-clusterip-service
        - name: MYSQL_PORT
          value: '3306'
        - name: MYSQL_ROOT_USER
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: username
```

Figure 27. Partial application deployment configuration file.

The volume to use for persistent storage is claimed by a separate object of type persistent volume claim named app-pvc. That volume is mounted to the specified

directory inside the container when the claim is fulfilled by Kubernetes master. The host and port environment variables used to connect to the development database running inside the cluster are targeted at the database cluster IP service that publishes ports to the deployment for internal cluster access.

Both the login credentials for authenticating with the private Azure registry and database login credentials are stored inside secrets that are created using imperative commands instead of applying configuration files. This is done in order to avoid storing sensitive data in plain text files. For added security the values inside secrets are stored in base64 encoded format. Secrets can be created from literal values passed as additional arguments from the command line. Alternatively, Kubectl supports generating secrets from pre-existing files. These commands vary depending on the type of the secret. One such common use case is for secrets that are of type docker-registry, which can have their values set directly using Docker's config.json file that stores registry authentication data.

Table 10. Kubectl imperative commands to create secrets.

Command	**Description**

`kubectl create secret generic mysecret \`
`  --from-literal=username=<username> \`
`  --from-literal=password=<password>`
	Imperative command to create a generic secret with two keys.

`kubectl create secret generic regsecret \`
`  --docker-server=<registry> \`
`  --docker-username=<username> \`
`  --docker-password=<password>`
	Imperative command to create a registry secret from literals.

`kubectl create secret generic regsecret \`
`  --from-file=.dockerconfigjson=<file> \`
`  --type=Kubernetes.io/dockerconfigjson`
	Imperative command to create a registry secret from file.

Once everything is in place all of the configuration files can be applied with Kubectl. Then the application can be tested in a local Kubernetes environment by entering Minikube's IP address into a web browser. Since the Nginx-Ingress is

serving traffic both over regular HTTP on port 80 and HTTPS on port 443, additional ports do not need to be specified.

If the ingress is not configured to use valid SSL certificates signed by a known CA, it will generate and use self-signed ones to serve HTTPS instead. These certificates should not be trusted by any web browser and will trigger a security warning that proceeding to the the site could pose threats. These warnings can be safely disregarded by continuing to the site.

4.11.3 Deployment with Helm

To make the Kubernetes application into a Helm chart, a new directory with the intended name of the application is created with the Chart.yaml, values.yaml files and a templates directory. All the application specific configuration files are copied under templates excluding those that are were created to test the Minikube development cluster such as the ingress. The Chart.yaml contains some basic metadata about the application that is displayed to other Helm users if the chart is hosted on a public Helm repository. In this case the chart is sent over directly, only the required fields are necessary.

```
apiVersion: v1
name: app
version: 1.0.0
appVersion: 1.0
description: Kubernetes Helm Chart for app
```

Figure 28. Helm Chart.yaml file.

The contents of the values.yaml file depend on which configuration options the vendor wishes to expose to the end user. Any values set inside the configuration files including container specific environmental variable declarations can be listed here. To link the values set in this file, the template configuration files must be edited to include a reference to the values. Helm uses Go’s built-in templating syntax for these references, which allow for a wide range of options, such as if statements and typecasting, among others.

Some thought should be given here on what the user may wish to change and what they can change without editing the contents of the image. Helm charts that are intended to be used by other developers can offer drastically different options than those intended for end users.

```yaml
namespace: default
replicaCount: 1
servicePort: 80
storageClaim: 1Gi

image:
    registry: someregistry.azurecr.io
    repository: app
    version: 1.0

application:
    domainUrl: some.domain.com
    rootDir: /var/www/app
    containerPort: 80

database:
    hostname: database-clusterip-service
    port: 3306
    user:
    password:
    name:
```

Figure 29. Helm values.yaml file.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: app-deployment
    namespace: {{ .Values.namespace }}
spec:
    replicas: {{ .Values.replicaCount }}
    selector:
```

Figure 30. Helm deployment template.

With these files in place, the user may simply enter their desired values into the values.yaml file, connect the service port to their own ingress and install the application via the helm chart. Kubernetes will use the included registry secret to authenticate with the provided image registry someregistry.azurecr.io and pull the

image app version 1.0 using Docker and run it as a container inside the deployment object that has been configured for it.

Installed charts and their versions are managed automatically. Helm provides an update function for updating the chart in the future and can keep track of the individual chart version info, enabling rolling back to previous versions as well.

Table 11. Helm commands.

Command	**Description**
`helm install my-app app/`	Install and name a Helm chart by providing a relative path.
`helm uninstall my-app`	Uninstall an installed Helm chart.
`helm upgrade my-app app/`	Upgrade an installed Helm chart by providing a relative path.
`helm rollback my-app 1.0`	Rollback an installed Helm chart to a previously installed version.
`helm delete --purge my-app`	Delete an installed Helm chart and all associated data.

With the Helm chart completed, this application can now be installed to any Kubernetes cluster simply by using the provided Helm chart directory totaling just under 4 Kilobytes in size. If the application were to be updated in the future, a new image version would be built and pushed to the registry. The image version and any other changes can then be edited in the values.yaml file and the updated Helm chart can be provided to the users.

As briefly mentioned in the previous chapter, Helm includes package manager functionality and production deployments and updates to several users can be au-

tomated by creating a Helm repository where the latest version of the chart is stored. Due to time constraints, this step had to be left out of the project.

5 CONCLUSION

Having finished this project, it is easy to see why containers are taking the world of software development by storm. Containers, backed up with an orchestrator such as Kubernetes and the vast infrastructure provided by cloud services, allow for easy deployments even to infrastructure owned and maintained by 3rd parties. While the necessary tools are not necessarily simple to adapt, in the end the pros outweigh the cons. Containerized deployments are easy to maintain and almost infinitely scalable across the web.

Docker is at the heart of all this and having a solid grasp of its workings is key to understanding the more complex structure of Kubernetes. The two are strikingly similar in many cases, adopting many identical patterns and methods of operation. At the surface Kubernetes is just a means to manage containerized applications. Its true power, however, stems from its intrinsic capabilities of unifying containerized systems across several physical computers and combining their individual resources into a single standardized operational unit. In this Docker is essential.

This synergy is what has enabled microservices to culminate from an ideal into a reality. Containers enable the separation of logic into smaller manageable parts for faster development while orchestrators speed up the deployment and updating cycles. Developers are also no longer restricted to using single programming language or a set of tools, but rather have the option to pick the best suited solutions for specific problems.

During the project it was highlighted that certain newer options are better suited for containerized environments, but legacy technologies benefit as well. Interpreted languages like PHP are fundamentally incapable of utilizing multiple computing cores for instance, but the replication of its entire runtime environment enables it to scale horizontally on modern multi-threaded systems.

All these changes that containers and microservices have brought forward have real world impact that translates to higher value for both software developers and end users alike.

The next steps to take from here would be to set up a Kubernetes cluster hosted on Azure Kubernetes Service and configure it for production following the rough outline set in this project. All the individual steps for developing and connecting containerized microservices are covered here on a basic level and may require additional refinement to be considered production ready.

What was not covered in this thesis was setting up a Helm repository for distributing Kubernetes deployments, using automated pipelines to deploy applications directly to the cluster, running cluster end-to-end tests, configuring the ingress for hosting applications and bridging traditional application development with the containerized model. All of these would be valid starting points for continuing the work that was laid out here.

REFERENCES

Docker Inc. 2020a. Why Docker? Referenced 2.5.2020.
https://www.docker.com/why-docker

Docker Inc. 2020b. Container Runtime. Referenced 2.5.2020.
https://www.docker.com/products/container-runtime

Docker Inc. 2020c. What is a Container? Referenced 2.5.2020.
https://www.docker.com/resources/what-container

Docker Inc. 2020d. Docker overview. Referenced 2.5.2020.
https://docs.docker.com/get-started/overview/

Docker Inc. 2020e. Dockerfile reference. Referenced 2.5.2020.
https://docs.docker.com/engine/reference/builder/

Docker Inc. 2020f. Best practices for writing Dockerfiles. Referenced 2.5.2020.
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Docker Inc. 2020g. Use Volumes. Referenced 2.5.2020.
https://docs.docker.com/storage/volumes/

Docker Inc. 2020h. Network Overview. Referenced 2.5.2020.
https://docs.docker.com/network/

Docker Inc. 2020i. Docker Hub Official images. Referenced 2.5.2020.
https://docs.docker.com/docker-hub/official_images/

Docker Inc. 2020j. Use the Docker command line. Referenced 2.5.2020.
https://docs.docker.com/engine/reference/commandline/cli/

Docker Inc. 2020k. Overview of Docker Compose. Referenced 2.5.2020.
https://docs.docker.com/compose/

Docker Inc. 2020l. Base OS images search results on Docker Hub. Referenced 2.5.2020. https://hub.docker.com/search?q=&type=image&image_filter=official&category=os

The Linux Foundation. 2020a. Borg: The Predecessor to Kubernetes. Referenced 2.5.2020. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/

The Linux Foundation. 2020b. Kubernetes Components. Referenced 4.5.2020. https://kubernetes.io/docs/concepts/overview/components/

The Linux Foundation. 2020c. Pod Overview. Referenced 4.5.2020. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

The Linux Foundation. 2020d. Deployments. Referenced 4.5.2020. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

The Linux Foundation. 2020e. Service. Referenced 4.5.2020. https://kubernetes.io/docs/concepts/services-networking/service/

The Linux Foundation. 2020f. Ingress. Referenced 5.5.2020. https://kubernetes.io/docs/concepts/services-networking/ingress/

The Linux Foundation. 2020g. Volumes. Referenced 5.5.2020. https://kubernetes.io/docs/concepts/storage/volumes/

The Linux Foundation. 2020h. Overview of kubectl. Referenced 5.5.2020. https://kubernetes.io/docs/reference/kubectl/overview/

The Linux Foundation. 2020i. Installing Kubernetes with Minikube. Referenced 6.5.2020. https://kubernetes.io/docs/setup/learning-environment/minikube/

Parikh, R. 2020. Kubernetes Helm: Why It Matters. Referenced 6.5.2020 https://platform9.com/blog/kubernetes-helm-why-it-matters/

Microsoft. 2020a. Azure Documentation: Containers. Referenced 6.5.2020. https://docs.microsoft.com/en-us/azure/?product=containers

Microsoft. 2020b. Introduction to private Docker registries in Azure. Referenced 6.5.2020. https://docs.microsoft.com/en-us/azure/container-registry/container-registry-intro

Microsoft. 2020c. What is Azure Pipelines. Referenced 7.5.2020. https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

Microsoft. 2020d. What is Azure CLI. Referenced 7.5.2020. https://docs.microsoft.com/en-us/cli/azure/what-is-azure-cli?view=azure-cli-latest